CONTRIBUTORS' GUIDE TO BRL-CAD

Table of contents

1. GETTING STARTED

A Call to Arms (and Contributors)

"The future exists first in the imagination, then in the will, then in reality." - *Mike Muuss*

Welcome to BRL-CAD! Whether you are a developer, documenter, graphic artist, academic, or someone who just wants to be involved in a unique open source project, BRL-CAD has a place for you. Our contributors come from all over the world and use their diverse backgrounds and talents to help maintain and enhance one of the oldest computer-aided design (CAD) packages used in government and industry today.

What is BRL-CAD?

BRL-CAD (pronounced *be-are-el-cad*) is a powerful, cross-platform, open source solid modeling system that includes interactive three-dimensional (3D) solid geometry editing, high-performance ray tracing support for rendering and geometric analysis, network-distributed framebuffer support, image and signal-processing tools, path tracing and photon mapping support for realistic image synthesis, a system performance analysis benchmark suite, an embedded scripting interface, and libraries for robust high-performance geometric representation and analysis.

For more than two decades, BRL-CAD has been the primary solid modeling CAD package used by the U.S. government to help model military systems. The package has also been used in a wide range of military, academic, and industrial applications, including the design and analysis of vehicles, mechanical parts, and architecture. Other uses have included radiation dose planning, medical visualization, terrain modeling, constructive solid geometry (CSG), modeling concepts, computer graphics education and system performance benchmark testing.

BRL-CAD supports a wide variety of geometric representations, including an extensive set of traditional implicit "primitive shapes" (such as boxes, ellipsoids, cones, and tori) as well as explicit primitives made from collections of uniform B-spline surfaces, non-uniform rational B-spline (NURBS) surfaces, n-manifold geometry (NMG), and purely faceted polygonal mesh geometry. All geometric objects may be combined using boolean set-theoretic CSG operations such as union, intersection and difference.

Overall, BRL-CAD contains more than 400 tools, utilities, and applications and has been designed to operate on many common operating system environments, including BSD, Linux, Solaris, Mac OS X, and Windows. The package is distributed in binary and source code form as Free Open Source Software (FOSS), provided under Open Source Initiative (OSI) approved license terms.

History and Vision

BRL-CAD was originally conceived and written by the late Michael Muuss, the inventor of the popular PING network program. In 1979, the U.S. Army Ballistic Research Laboratory (BRL) (the agency responsible for creating ENIAC, the world's first general-purpose electronic computer in the 1940s) identified a need for tools that could assist with the computer simulations and analysis of combat vehicle systems and environments. When no existing CAD package was found to be adequate for this specialized purpose, Mike and fellow software developers began developing and assembling a unique suite of utilities capable of interactively displaying, editing, and interrogating geometric models. Those early efforts subsequently became the foundation on which BRL-CAD was built.

Development of BRL-CAD as a unified software package began in 1983, and its first public release came in 1984. Then, in 2004, BRL-CAD was converted from a limited-distribution U.S. government-controlled code to an open source project, with portions licensed under the LGPL and BSD licenses.

Today, the package's source code repository is credited as being the world's oldest, continuously developed open source repository. As a project, pride is taken in preserving all history and contributions.

The ongoing vision for BRL-CAD development is to provide a robust, powerful, flexible, and comprehensive solid modeling system that includes:

- Faithful high-performance geometric representation.
- Efficient and intuitive geometry editing.
- Comprehensive conversion support for all solid geometry formats.
- Effective geometric analysis tools for 3D CAD.

Key Strengths

All CAD packages are not alike. Among the many strengths of the BRL-CAD package are the following:

- BRL-CAD is **open source**! Don't like something? You can make it better.
- You can leverage **decades of invested development**. BRL-CAD is the most feature-filled open source CAD system available, with hundreds of years time invested.
- **Your work will get used**. BRL-CAD is in production use and downloaded thousands of times every month by people all around the world.
- You have the ability to create extensively **detailed realistic models**.
- You can model objects on scales ranging from (potentially) the subatomic through the galactic, while essentially providing **all the details, all the time**.
- You can leverage **one of the fastest** raytracers in existence (for many types of geometry).
- You can convert to and from a wide range of **geometry file formats**.
- BRL-CAD has a powerful, **customizable scripting interface** with many advanced editing and processing capabilities.

Want to Be a Contributor?

With BRL-CAD being a part of the open source community since 2004, contributors from all over the world are able to enhance the features and functions of the package in many different ways. In return, these contributors have had a unique opportunity to:

- Join a team of passionate and talented contributors who share the common values of open source development. Open source emphasizes free redistribution; openly available source code; full, open participation; and nondescrimination against individuals, groups, technologies, or fields of interest. (To learn more, see http://opensource.org.)
- Drive needed improvements in the open source software community's support for solid modeling and CAD software capabilities.
- Experiment with new and state-of-the-art algorithms and ideas within the context of a fully open CAD system that is in production use and has an established user community.
- Become a better developer. Whether you're a newbie or a seasoned developer with decades of experience, you can always work on a BRL-CAD project that is catered toward improving your abilities.

- Become part of a legacy. Participate in a robust and historically significant open source project that goes all the way back to the days of the DEC PDP-11/70 and VAX-11/780.
- Gain practical experience working on a real-world, large-scale software project.

If you would like to be a BRL-CAD contributor, the primary areas currently identified for future development and enhancement include the following:

- **Improved graphical user interface and usability** to accommodate increasingly varied user needs and participation levels. This includes improving the look-and-feel and features of:
    - the primary editing graphical interface (MGED)
    - the geometric visualization and interaction management system (libdm).
- **Improved hybrid boundary representation geometry support** to support all 3D CAD models regardless of whether they use implicit or explicit geometric representation. Geometry formats we are particularly focusing on include:
    - volumetric models (VOL)
    - spline-surface (for example, NURBS) and polygonal (for example, triangle mesh) boundary representations (BREP)
    - implicit primitives.
- **Improved geometry services and functionality**, including the ability to provide:
    - multiuser access controls
    - comprehensive revision history
    - collaborative enhanced multiuser modeling
    - more flexible application development.

In addition, BRL-CAD's existing geometry kernel functions are continuously being refactored. Help turn them into a comprehensive, scriptable command framework, create an object-oriented geometry kernel application programming interface (API), or build a lightweight network daemon protocol for language agnostic client application development.

- **Improved open source awareness and increased development participation** via:
    - establishing strong open source community ties
    - improving software documentation
    - openly welcoming new contributors and users.

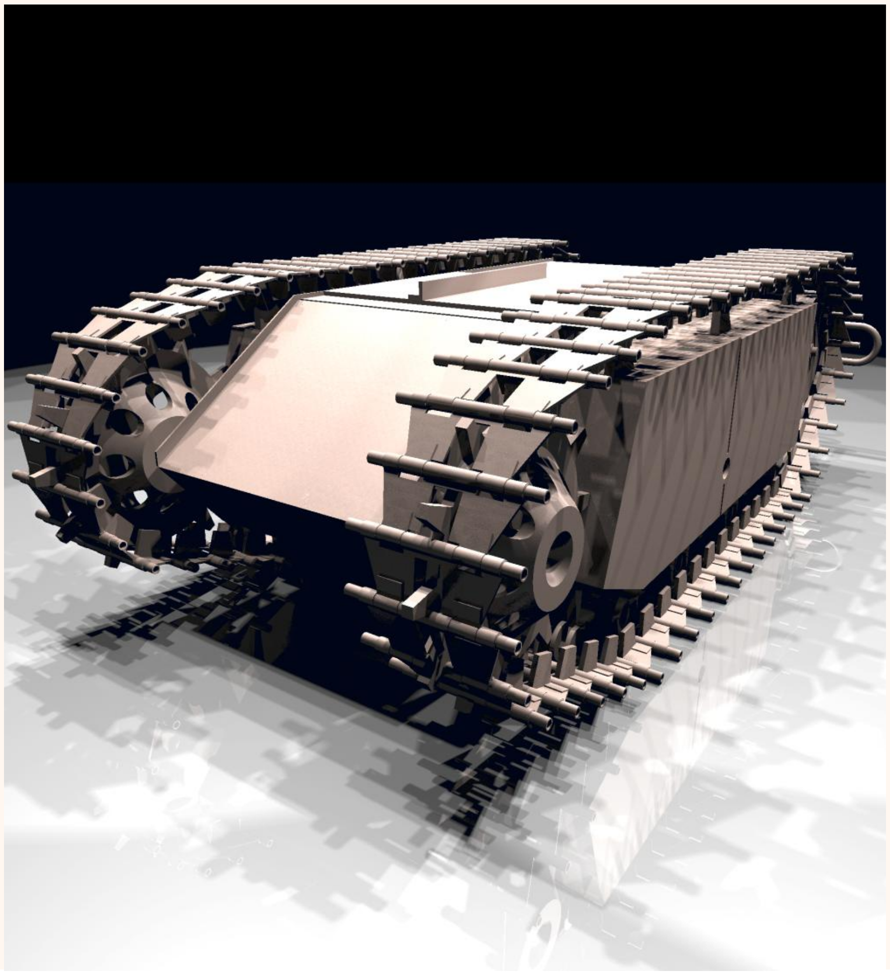
Let the contributions begin!

Feature Overview

BRL-CAD has thousands of distinct features that have been developed over a number of decades. One strength of a solid modeling system with integrated high-performance rendering is the ability to showcase some of those features graphically.

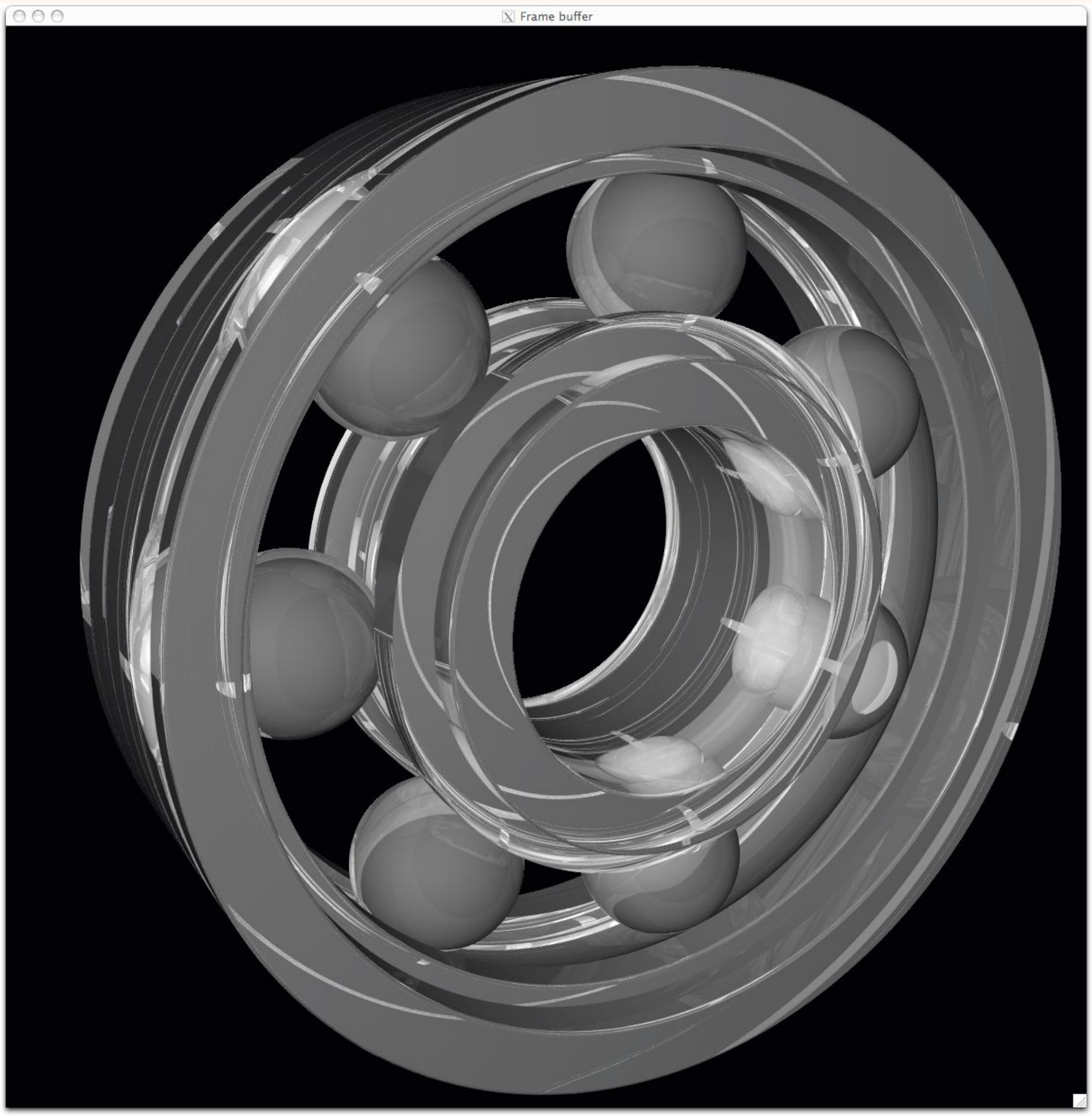
Let's take a quick look at just some of the high-level features provided by BRL-CAD.

Solid Geometry



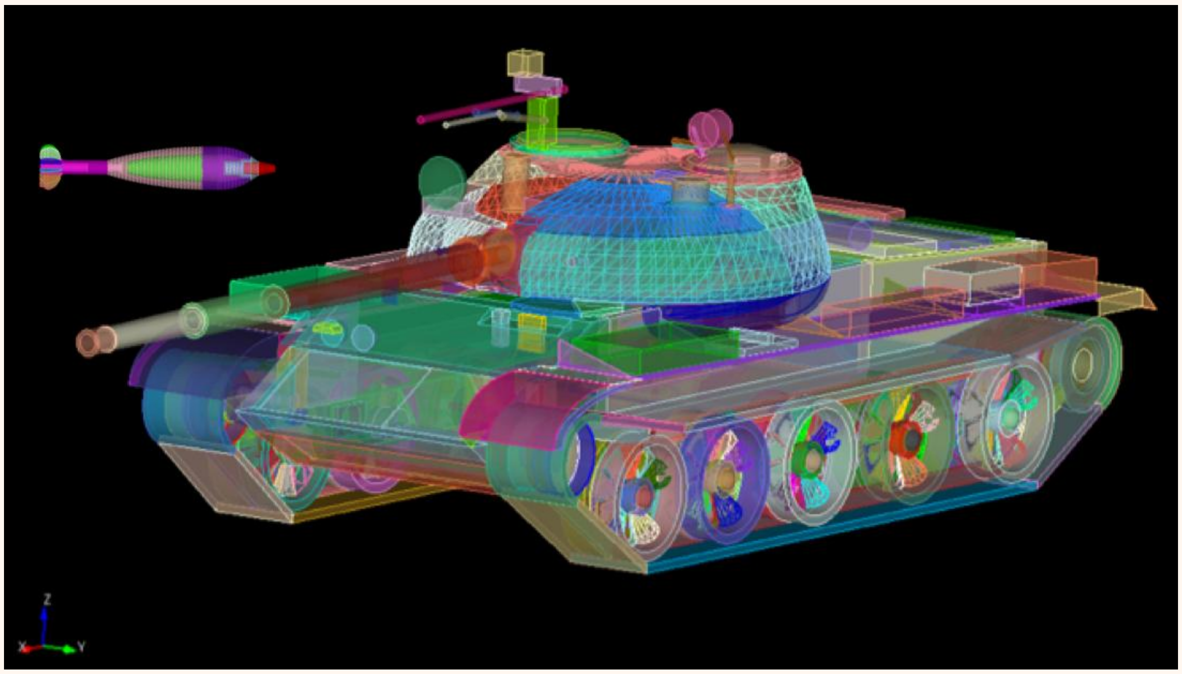
BRL-CAD focuses on solid modeling CAD. Solid modeling is distinguished from other forms of geometric modeling by an emphasis on being physically accurate, fully describing 3D space. Shown is a 3D model of a Goliath tracked mine, a German-engineered remote controlled vehicle used during World War II. This model was created by students new to BRL-CAD in the span of about 2 weeks, starting from actual measurements in a museum.

Raytracing

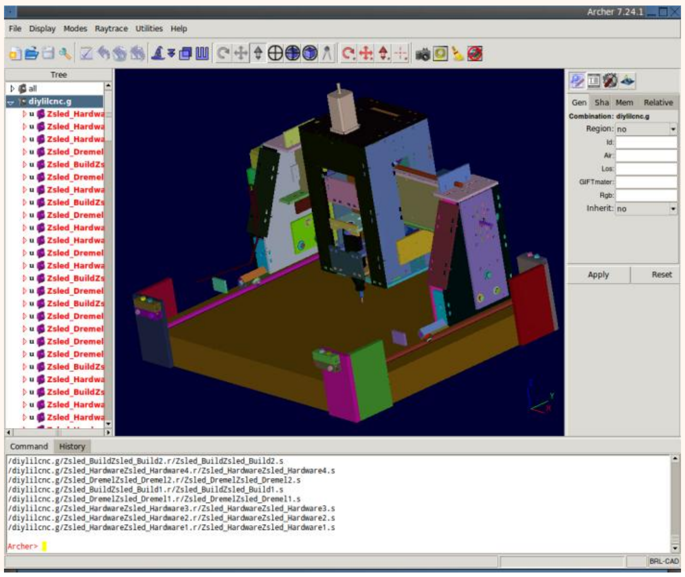


Raytracing is central to BRL-CAD as a means for performing geometric analysis (e.g., calculating weights and moments of inertia) and for rendering images for visualization purposes. The image shown is a BRL-CAD 2D framebuffer screenshot displaying the rendering of a ball bearing. The bearing is modeled with a material appearance resembling acrylic glass, and this raytracing result shows reflection, refraction, shadowing, and some caustic effects.

Geometry Conversion



As shown, a BRL-CAD target description can be converted to a finite element mesh (FEM) using the BRL-CAD g-sat exporter and Cubit from Sandia National Laboratories.



```
/diylilcnc.g/Zsled_BuildZsled_Build2.r/Zsled_BuildZsled_Build2.s
/diylilcnc.g/Zsled_HardwareZsled_Hardware4.r/Zsled_HardwareZsled_Hardware4.s
/diylilcnc.g/Zsled_DremelZsled_Dremel2.r/Zsled_DremelZsled_Dremel2.s
/diylilcnc.g/Zsled_BuildZsled_Build1.r/Zsled_BuildZsled_Build1.s
/diylilcnc.g/Zsled_DremelZsled_Dremel1.r/Zsled_DremelZsled_Dremel1.s
/diylilcnc.g/Zsled_HardwareZsled_Hardware3.r/Zsled_HardwareZsled_Hardware3.s
/diylilcnc.g/Zsled_HardwareZsled_Hardware2.r/Zsled_HardwareZsled_Hardware2.s
/diylilcnc.g/Zsled_HardwareZsled_Hardware1.r/Zsled_HardwareZsled_Hardware1.s

Archer>
```

This screenshot shows a model imported from the Rhino3D 3DM file format into BRL-CAD as NURBS boundary representation geometry, visualized via OpenGL.

More Cowbell

Not all of BRL-CAD's capabilities lend themselves well to pretty pictures, but some are definitely worth mentioning. Among the thousands of features in BRL-CAD, here are some additional capabilities that are central to our project ethos.

Geometric Analysis

A particular strength of the BRL-CAD software lies in its ability to build and analyze realistic models of complex objects. There are a number of features aimed at inspecting, preparing, verifying, and validating geometry models. Single-ray sampling can be used for measuring thicknesses or distances, and certain 3D analyses are possible (such as calculating volume, centroids, and moments of inertia). BRL-CAD also has numerous facilities for detecting and resolving assembly or part interferences where two objects spatially overlap each other.

High-Performance Design

BRL-CAD is designed from the ground up with performance in mind. Considerable attention has been put into in-memory and on-disk data storage efficiency. BRL-CAD is capable of handling complex geometry models that are often impossible to open with other systems without changing hardware requirements. BRL-CAD's ray tracing infrastructure is one of the fastest in the world for implicit geometry representations and is continually seeking performance advancements for other explicit representation types, such as polygonal mesh geometry and NURBS surface models. BRL-CAD's distributed ray tracing support is recognized as the world's first "real-time" ray tracing implementation, achieving several frames per second in the 1980s.

Symmetric Multi-Processing

BRL-CAD efficiently leverages symmetric multi-processing (SMP) capabilities of desktop, server, and supercomputing systems, where an arbitrary number of processing cores can be put to work on a computational task. BRL-CAD's ray tracing library is commonly leveraged for performing highly detailed geometric analysis, driving third-party simulations, and producing animations.

Modular Architecture

As a large software package developed over a relatively long period of time, BRL-CAD has necessarily been designed and evolved with modularity in mind. Functionality is implemented across hundreds of application modules, commands, and libraries designed to work together. Hundreds of application binaries work together supporting efficient customizable workflows. Core geometry editing capabilities are implemented as commands that can be easily extended, replaced, or improved upon. All functionality and features are built on top of a core set of libraries that encapsulate common capabilities. One of the best ways to get involved is to add a new module or improve an existing one.

Cross-Platform Portability

BRL-CAD has an extensive history of investment in and attention toward cross-platform portability. This heritage includes systems such as a DEC VAX-11/780 running 4.3 BSD, DECStations running ULTRIX, Silicon Graphics machines running IRIX, Cray supercomputers running UNICOS, and so much more. Today, BRL-CAD's hardware support includes everything from minimal laptops and desktops to gigantic distributed supercomputers. And it is commonly run on Linux, Windows, Mac OS X, BSD, Haiku, Solaris, and other desktop operating systems. We aim to be "embarrassingly portable."

ISO STEP 10303

STandard for the Exchange of Product Model Data (STEP) is an ISO standard describing a product's full life cycle. One small portion of that gigantic standard describes a complex geometry file format that fortunately has been adopted by most commercial CAD systems. BRL-CAD is proud to be one of the few open source software systems that is able to read and write STEP geometry files.

Performance Benchmark

The BRL-CAD Benchmark provides a practical metric of real-world performance. Correlated with a longstanding heritage of providing verifiable and repeatable behavior throughout the package, the Benchmark compares a given compilation's ray tracing performance against the results from one of the very first systems to support BRL-CAD: a VAX 11/780 running BSD. The mathematically intensive computations exercise the processing unit, system memory, various levels of data and instruction cache, the operating system, thread concurrency efficiency, data coherency, and compiler optimization capabilities. The performance results let you weigh the relative computational strength of a given platform. With the right controls in place, the Benchmark can tell you whether a given operating system is more efficient than another, whether a particular compiler really makes a difference, or just how much of an improvement a particular piece of hardware provides. We have results tracing back several decades of computing.

2. DEVELOPERS

Working with Our Code

BRL-CAD consists of more than 1 million lines of source code spanning more than 20 foundation libraries and 400 application modules.

The majority of BRL-CAD is written in highly portable C and C++, with some GUI and scripting components written in Tcl/Tk[1]. There is also some support for, and bindings to, other languages available. POSIX[2] shell scripts are used for deployment integration testing. BRL-CAD uses the CMake[3] build system for compilation and unit testing.

The Big Picture

The source code and most project data are stored in a Subversion[4] version control system for change tracking and collaborative development. Trunk development is generally stable, but cross-platform compilation is not guaranteed. A separate branch (named *STABLE*) provides a higher level of quality assurance. Every released version of BRL-CAD is tested and tagged.

The project aims for an **It Just Works** approach to compilation whereby a functional build of BRL-CAD is possible without needing to install more than a compiler, CMake, and a build environment--for example, GNU Make or Microsoft Visual Studio. BRL-CAD provides all of the necessary third-party dependencies for download and compilation convenience within source distributions but by default will build using system versions of those dependencies if available.

As with any large system that has been under development for a number of years, there are vast sections of code that may be unfamiliar, uninteresting, or even daunting.

Don't panic. BRL-CAD has been intentionally designed with layering and modularity in mind.

You can generally focus in on the enhancement or change that interests you without being too concerned with other portions of the code. You should, however, do some basic research to make sure what you plan to contribute isn't already in the BRL-CAD code base.

History of the Code

As mentioned previously, the initial architecture and design of BRL-CAD began in 1979. Development as a unified package began in 1983. The first public release was in 1984. And on December 21, 2004, BRL-CAD became an open source project[5].

BRL-CAD is a mature code base that has remained active over decades due to continual attention on design and maintainability. Since the project's inception, more than 200 people have directly contributed to BRL-CAD. The project has historically received support from numerous organizations within academia, commercial industry, various government agencies, and from various independent contributors. We credit all contributors in BRL-CAD's authorship documentation[6].

The following diagram illustrates how the number of lines of code in BRL-CAD has

changed over time:

System Architecture

BRL-CAD is designed based on a UNIX[7] methodology of the command-line services, providing many tools that work in harmony to complete a specific task. These tools include geometry and image converters, signal and image processing tools, various raytrace applications, geometry manipulators, and much more.

To support what has grown into a relatively large software system, BRL-CAD takes advantage of a variety of support libraries that encapsulate and simplify application development. At the heart of BRL-CAD is a multi-representation ray tracing library named LIBRT. BRL-CAD specifies its own file format (files with the extension .g or .asc) for storing information on disk. The ray tracing library uses a suite of other libraries for other basic application functionality.

Tenets of Good Software

BRL-CAD's architecture is designed to be as cross-platform and portable as is realistically and reasonably possible. As such, BRL-CAD maintains support for many legacy systems and devices provided that maintaining such support is not a significant burden on new development.

The code adheres to a published change deprecation and obsolescence policy[8] whereby features that have been made publicly available are not removed without appropriate notification. Generally there should be a compelling motivation to remove any existing functionality, but improvements are encouraged.

BRL-CAD has a longstanding heritage of maintaining verifiable, validated, and repeatable results in critical portions of the package, particularly in the ray tracing library. BRL-CAD includes regression tests that will compare runtime behavior against known results and report any deviations from previous results as failures. Considerable attention is put into verification and validation throughout BRL-CAD. *Incorrect* behavior does not need to be preserved simply to maintain consistency, but it is rare to find genuine errors in the baseline testing results. So, anyone proposing such a behavior change will have to conclusively demonstrate that the previous result is incorrect.

Code Layout

The basic layout of BRL-CAD's source code places public API headers in the top-level *include* directory and source code for both applications and libraries in the *src* directory. The following is a partial listing of how the code is organized in a checkout or source distribution. Note that some subdirectories contain a README file with more details on the content in that directory.

Applications & Resources

db/

- Example geometry

doc/

- Project documentation

doc/docbook

- User documentation in XML format
- See doc/docbook/README for more details

include/

- **Public API headers**

regress/
- Scripts and resources for regression testing

src/
- **Application and library source code**
- **See src/README for more details**

src/conv/
- Geometry converters

src/fb/
- Tools for displaying data in windows

src/mged/
- Main GUI application: Multi-device Geometry EDitor

src/other/
- 3rd party frameworks (Tcl/Tk, libpng, zlib, etc.)

src/proc-db/
- Examples on creating models programmatically

src/rt*/
- Ray tracing applications

src/util/
- Image processing utilities

Libraries

src/libbn/
- Numerics library: vector/matrix math, random number generators, polynomial math, root solving, noise functions, and more

src/libbu
- Utility library: string handling, logging, threading, memory management, argument processing, container data structures, and more

src/libgcv/
- Geometry conversion library for importing or exporting geometry in various formats

src/libged/
- Geometry editing library containing the majority of our command API

src/libicv/
- Image conversion library for importing, processing, and exporting image data

src/libpkg/
- Network "package" library for basic client-server communication

src/librt/

- Ray tracing library including routines for reading, processing, and writing geometry

src/libwdb/
- Simple (write-only) library for creating geometry

src/lib*/tests/
- API Unit tests

Code Conventions

BRL-CAD has a *STABLE* branch in SVN that should always compile and run on all supported platforms. The primary development branch *trunk*, unlike STABLE, is generally expected to compile but may occasionally fail to do so during active development.

Languages

The majority of BRL-CAD is written in ANSI/POSIX C with the intent of strictly conforming with the C standard. The core libraries are all C API, though several--such as the LIBBU and LIBRT libraries--use C++ for implementation details. Our C libraries can use C++ for implementation detail, but they cannot expose C++ in the public API.

Major components of the system are written in the following languages:

- STEP and NURBS boundary representation support: C++
- The MGED geometry editor: a combination of C, Tcl/Tk, and Incr Tcl/Tk
- The BRL-CAD Benchmark, build system, and utility scripts: POSIX-compliant Bourne Shell Script
- Initial implementation of a BRL-CAD Geometry Server: PHP

Source code files use the following extensions:

- C files: .c
- Header files: .h
- C++ files: .cpp
- PHP files: .php
- Tcl/Tk files: .tcl or .tk
- POSIX Bourne-style shell scripts: .sh
- Perl files: .pl (program) or .pm (module)

With release 7.0, BRL-CAD has moved forward and worked toward making all of the software's C code conform strictly with the ANSI/ISO standard for C language compilation (ISO/IEC 9899:1990, or c89). Support for older compilers and older K&R-based system facilities is being migrated to build system declarations or preprocessor defines, or is being removed outright. You can, however, make modifications that assume compiler conformance with the ANSI C standard (c89).

Coding Style

To ensure consistency, the coherence of the project, and the long-term maintainability of BRL-CAD, we use a defined coding style and conventions that contributors are expected to follow. Our coding style is documented in the HACKING file of any source distribution.

Our style may not be your preferred style. While we welcome discussion, we will always prefer consistency over any personal preference. Contributions that do not follow our style will generally be rejected until they do.

Here are some highlights of our style:

- Global variables, structures, classes, and other public data containers are discouraged within application code. Do not add any new global variables to existing libraries. Global variables are often a quick solution to some deeper coding problem. However, they carry significant maintenance costs, introduce complexity to the code, make multi-threading support more costly, pollute the public API (symbol-wise at a minimum), increase security risks, are error-prone to use, and usually complicate future efforts to refactor and restructure the code. Using static variables (whether function- or static/file-scoped) is a viable alternative. Restructuring the logic to not be stateful is even better.
- Exact floating point comparisons are unreliable without requiring IEEE-compliant floating point math, but BRL-CAD does not require such math for portability and for performance reasons. When floating point comparisons are necessary, use the NEAR_EQUAL and NEAR_ZERO macros with a specified tolerance or the EQUAL and ZERO macros where a tolerance is indeterminate. All the macros are available by including bn.h, part of libbn.
- The code should strive to achieve conformance with the GNU coding standard with a few exceptions. One such exception is **not** using the GNU indentation style, but instead using the BSD KNF indentation style, which is basically the K&R indentation style with character indentation consistent with the file that you're editing. If this is confusing, use spaces to indent and run the sh/ws.sh script to convert spaces to tabs. We value consistency to preserve maintainability.
- Stylistic whitespace

  - No space immediately inside parentheses.

```
while (1) { ...                  /* ok */
for (i = 0; i < max; i++) { ...   /* ok */
while ( max ) { ...              /* discouraged */
```

- Commas and semicolons are followed by whitespace.

```
int main(int argc, char *argv[]); /* ok */
for (i = 0; i < max; i++) { ...   /* ok */
```

- No space on arrow operators.

```
structure->member = 5;          /* ok */
structure -> member = 5;        /* bad */
```

- Native language statements (if, while, for, switch, and return) have a separating space; functions do not.

```
int my_function(int i);         /* ok, no space */
while (argc--) ...              /* ok, has space */
if( var == val )                /* discouraged */
switch(foo) ...                 /* discouraged */
```

- Comments should have an interior space and be without tabs.

```
/** good single-line doxygen */
/* good */
/*bad*/
/*    discouraged */
/*  discouraged  */

/**
 * good:
 * multiple-line doxygen comment
 */
```

- Naming symbols

Variable and public API function names should almost always begin with a lowercase letter.

```
double localVariable; /* ok */

double LocalVariable; /* bad (looks like class or    constructor) */

double _localVar;     /* bad (looks like member variable)      */
```

Do not use Hungarian notation or its variations to show the type of a variable. An exception can be made for pointers on occasion. The name should be concise and meaningful--typing a descriptive name is preferred to someone spending time trying to learn what the name of the variable means.

```
char *name;    /* ok  */
char *pName;   /* discouraged for new code, but okay */

char *fooPtr;  /* bad */

char *lpszFoo; /* bad */
```

Constants should be all upper-case with word boundaries optionally separated by underscores.

```
static const int MAX_READ = 2;  /* ok  */
static const int arraySize = 8; /* bad */
```

Public API (global) function names should be in lowercase with underscores to separate words. Most functions within the core libraries are named with the following convention: [library]_[group]_[action]

```
bu_vls_strcat()
bn_mat_transpose()
```

Naming exceptions are allowed where the API intentionally mirrors some other familiar programming construct--for example, bu_malloc()+bu_free())--but be as consistent as possible within a file and across a library's API.

- BRL-CAD uses *The One True Brace Style* from BSD KNF and K&R[9]. Opening braces should be on the same line as their statement; closing braces should line up with that same statement. Functions, however, are treated specially, and we place their opening braces on separate lines.

```
static int
 some_function(char *j)
 {
    for (i = 0; i < 100; i++) {
        if (i % 10 == 0) {
            j += 1;
        } else {
            j -= 1;
        }
    }
 }
```

1. http://www.tcl.tk/^
2. http://en.wikipedia.org/wiki/POSIX^
3. http://www.cmake.org/^
4. http://subversion.apache.org/^
5. http://developers.slashdot.org/story/05/01/08/1823248/us-army-research-lab-opens-brl-cad-source^
6. See the AUTHORS file in a source distribution.^
7. http://en.wikipedia.org/wiki/Unix^
8. See the CHANGES file in a source distribution.^
9. http://en.wikipedia.org/wiki/Indent_style^

4. Appendix: Resources and Examples

Example Code: Root Solving

Root solving is (among other things) a key step in the raytracing of many of BRL-CAD's primitives. The following examples illustrate how to solve various types of polynomial equations using BRL-CAD's root solver.

```
#include "common.h"

#include "bu.h"

#include "vmath.h"

#include "bn.h"

#include "raytrace.h"

int

main(int argc, char *argv[])

{

   bn_poly_t equation; /* holds our polynomial equation */

   bn_complex_t roots[BN_MAX_POLY_DEGREE]; /* stash up to six roots */

   int num_roots;


   if (argc > 1)

       bu_exit(1, "%s: unexpected argument(s)\n", argv[0]);

   /*********************************************

    * Linear polynomial (1st degree equation):
```

```
 *  A*X + B = 0
 * [0]   [1]     <= coefficients
 */
equation.dgr = 1;
equation.cf[0] = 1;  /* A */
equation.cf[1] = -2;  /* B */


/* print the equation */
bu_log("\n*** LINEAR ***\n");
bn_pr_poly("Solving for Linear", &equation);
/* solve for the roots */
num_roots = rt_poly_roots(&equation, roots, "My Linear Polynomial");
if (num_roots == 0) {
    bu_log("No roots found!\n");
    return 0;
} else if (num_roots < 0) {
    bu_log("The root solver failed to converge on a solution\n");
    return 1;
}


/*  A*X + B = 0
```

```
 *  1*X + -2 = 0
 *    X -  2 = 0
 *    X      = 2
 */
/* print the roots */
bu_log("The root should be 2\n");
bn_pr_roots("My Linear Polynomial", roots, num_roots);
/*********************************************
 * Quadratic polynomial (2nd degree equation):
 *  A*X^2 + B*X + C = 0
 * [0]     [1]   [2]     <=coefficients
 */
equation.dgr = 2;
equation.cf[0] = 1;  /* A */
equation.cf[1] = 0;  /* B */
equation.cf[2] = -4; /* C */

/* print the equation */
bu_log("\n*** QUADRATIC ***\n");
bn_pr_poly("Solving for Quadratic", &equation);
```

```
/* solve for the roots */

num_roots = rt_poly_roots(&equation, roots, "My Quadratic Polynomial");

if (num_roots == 0) {

    bu_log("No roots found!\n");

    return 0;

} else if (num_roots < 0) {

    bu_log("The root solver failed to converge on a solution\n");

    return 1;

}


/*  A*X^2 + B*X +  C = 0

 *  1*X^2 + 0*X + -4 = 0

 *    X^2       -  4 = 0

 * (X - 2) * (X + 2) = 0

 *  X - 2 = 0, X + 2 = 0

 *  X = 2, X = -2

 */

/* print the roots */

bu_log("The roots should be 2 and -2\n");

bn_pr_roots("My Quadratic Polynomial", roots, num_roots);

/****************************************
```

```
 * Cubic polynomial (3rd degree equation):
 *  A*X^3 + B*X^2 + C*X + D = 0
 * [0]     [1]     [2]   [3]     <=coefficients
 */
equation.dgr = 3;
equation.cf[0] = 45;
equation.cf[1] = 24;
equation.cf[2] = -7;
equation.cf[3] = -2;

/* print the equation */
bu_log("\n*** CUBIC ***\n");
bn_pr_poly("Solving for Cubic", &equation);

/* solve for the roots */
num_roots = rt_poly_roots(&equation, roots, "My Cubic Polynomial");
if (num_roots == 0) {
   bu_log("No roots found!\n");
   return 0;
} else if (num_roots < 0) {
   bu_log("The root solver failed to converge on a solution\n");
```

```
    return 1;
 }


 /* print the roots */
 bu_log("The roots should be 1/3, -1/5, and -2/3\n");
 bn_pr_roots("My Cubic Polynomial", roots, num_roots);
 /*******************************************
  * Quartic polynomial (4th degree equation):
  *  A*X^4 + B*X^3 + C*X^2 + D*X + E = 0
  * [0]    [1]    [2]    [3]   [4]    <=coefficients
  */
 equation.dgr = 4;
 equation.cf[0] = 2;
 equation.cf[1] = 4;
 equation.cf[2] = -26;
 equation.cf[3] = -28;
 equation.cf[4] = 48;


 /* print the equation */
 bu_log("\n*** QUARTIC ***\n");
 bn_pr_poly("Solving for Quartic", &equation);
```

```
/* solve for the roots */

num_roots = rt_poly_roots(&equation, roots, "My Quartic Polynomial");

if (num_roots == 0) {

   bu_log("No roots found!\n");

   return 0;

} else if (num_roots < 0) {

   bu_log("The root solver failed to converge on a solution\n");

   return 1;

}

/* print the roots */

bu_log("The roots should be 3, 1, -2, -4\n");

bn_pr_roots("My Quartic Polynomial", roots, num_roots);

/*******************************************

 * Sextic polynomial (6th degree equation):

 *  A*X^6 + B*X^5 + C*X^4 + D*X^3 + E*X^2 + F*X + G = 0

 * [0]    [1]    [2]    [3]    [4]    [5]   [6]  <=coefficients

 */


equation.dgr = 6;

equation.cf[0] = 1;
equation.cf[1] = -8;

equation.cf[2] = 32;

equation.cf[3] = -78;

equation.cf[4] = 121;

equation.cf[5] = -110;

equation.cf[6] = 50;


/* print the equation */

bu_log("\n*** SEXTIC ***\n");

bn_pr_poly("Solving for Sextic", &equation);


/* solve for the roots */

num_roots = rt_poly_roots(&equation, roots, "My Sextic Polynomial");

if (num_roots == 0) {

   bu_log("No roots found!\n");

   return 0;

} else if (num_roots < 0) {

   bu_log("The root solver failed to converge on a solution\n");

   return 1;

}
```

```
    /* print the roots */

    bu_log("The roots should be 1 - i, 1 + i, 2 - i,2 + i, 1 - 2*i, 1 + 2*i \n");

    bn_pr_roots("My Sextic Polynomial", roots, num_roots);



    return 0;

}
```